# Towards a Closed-Looped Automation for Service Assurance with the DXAGENT

Korian Edeline, Thomas Carlisi, Justin Iurman
*Montefiore Institute*
*Université de Liège*
Liège, Belgium

Benoît Claise
*Intelligent Operations & Management*
*Huawei*
Liège, Belgium

Benoit Donnet
*Montefiore Institute*
*Université de Liège*
Liège, Belgium

***Abstract*—Recently, *Intent-Based Networking* (IBN) has known an increasing interest from both the industry and research communities. IBN comes with the advantage of easily expressing the desired state of a network. In parallel, service assurance, through observability, has been becoming more prevalent to maximize the business continuity. In that spirit, *Service Assurance in Intent-based Networking* (SAIN), is in the process of standardization at the IETF and proposes a general framework towards closed-loop automation for service assurance.**

**This paper introduces the Diagnostic Agent (DXAGENT), an open-source SAIN Agent whose purpose is to determine symptoms and health levels of the different subservices of a network service. As such, the DXAGENT appears as a first step towards closed-loop automation for service assurance. This paper describes the DXAGENT implementation and demonstrates its efficiency through use cases.**

***Index Terms*—Intent-Based Networking; SAIN; DXAGENT; closed-loop automation;**

## I. INTRODUCTION

*Intent Based Networking* (or IBN) [1], [2], [3] has been recently introduced to allow a network administrator to formulate the desired state of the network in a high-level manner and have the network orchestration done automatically. The IBN market is fast growing as its size exceeded USD900 millions in 2019 and is estimated to grow to USD4.5 billions by 2026 [4].

In parallel to the rise of IBN, the last fifteen years have witnessed a strong evolution of the Internet. from a hierarchical, relatively sparsely interconnected network to a flatter and much more densely inter-connected network [5], [6], [7] in which hyper giant distribution networks (HGDNs, - e.g., Facebook, Google, Netflix) are responsible for a large portion of the world traffic [8]. HGDNs are becoming the de-facto main actors of the modern Internet. The very same set of actors have fueled the move to very large data center networks (DCNs), along with the evolution to cloud native networking. For those networks, the Command Line Interface (CLI) is no longer the norm for configuring their networks. Indeed, networks must be automated and data model-driven management, with more and more data models, from which APIs are generated, being standardized. The Yang catalog[1] follows that example.

This work has been funded by a Cisco grant CG# 1717376, a Cisco grant CG# 2713379 and CyberExcellence project funded by the Walloon Region, under number 2110186.

[1]See https://yangcatalog.org/private-page

In addition to configuration automation, the networking industry has been adopting (model-driven) *observability* as a compulsory step to stream any monitoring information to a collector [9], [10]. However, there are challenges with the data collection, transport, analysis, and performance in scaled-out implementations of networked devices with high volumes of telemetry. Collecting all the observability information into a centralized manner in order to analyze all information to find the network degradation root cause is certainly a possibility but suffer from the "finding a needle in the haystack" issue, on top of being reactive. The assurance should be done closer to the network and domain per domain in order to scale and be efficient.

This paper focuses on the issue of using telemetry as a closed loop mechanism for service assurance, as a foundational step towards IBN. In particular, we follow the *Service Assurance for Intent-based Networking Architecture* (SAIN) [11] for developing a SAIN Agent responsible for building an assurance graph and detect symptoms based on metrics collected on the device and in the network. Our SAIN Agent is called the *Diagnostic Agent* (DXAGENT) [12]. To the best of our knowledge, the DXAGENT is the first open-source implementation of a SAIN Agent and can be used as a building block towards closed-loop automation for service assurance.

Our DXAGENT is able to retrieve data from multiple sources, on the machine on which it runs itself (e.g., CPU information, memory information) but also on other devices on the network thanks to telemetry observation [13], [14], [15]. Next, those observations are normalized and the various subservices are discovered in order to build the assurance graph. As a last step, the DXAGENT checks for potential symptoms based on user-defined rules applied to the normalized metrics and propagates the corresponding health scores along the subservice assurance graph. This paper describes the DXAGENT implementation and demonstrates its usefulness through use cases. All our implementation is open-source and freely-available [16].

The remainder of this paper is organized as follows: Sec. II discusses the required background for this paper, focusing on the SAIN framework; Sec. III describes our DXAGENT implementation; Sec. IV introduces use cases that demonstrate the DXAGENT efficiency; finally, Sec. V concludes this paper by summarizing its main achievements.

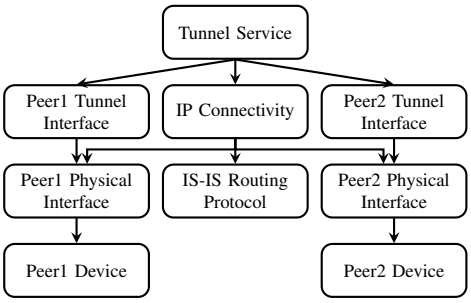



Fig. 1. Assurance graph – tunnel example [11].

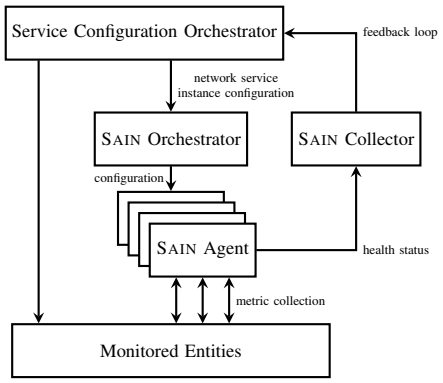



Fig. 2. SAIN architecture.

## II. SAIN

The *Service Assurance for Intent-based Networking Architecture* (SAIN) [11] is a generic architecture developed by the IETF for assuring that service instances are running correctly. In particular, it allows to answer where is the fault when a service degrades, focusing on the symptoms and the root cause. Further, when a network component fails, it helps pointing the impacted services. Those services are provided by SAIN by, first, decomposing the problem into smaller components, i.e., the *subservices*. An assurance graph (typically a directed acyclic graph) is built to link those subservices in order to map service intent. In particular, the root of the graph represents the service to be assured, while the children represent the subservices directly dependent on the service. The different subservices are assured independently and a service health score is possibly inferred.

Fig. 1 shows an example of assurance graph for a tunnel service: the tunnel service depends on the two peer interfaces of the tunnel but also on an IP connectivity that, itself, depends on its routing protocol. We can add to this that the peers interfaces depend on their physical interfaces, themselves depending on the devices behind the interface.

One main principle of the SAIN architecture is to maintain a correct assurance graph despite possible changes in services or network conditions. The SAIN framework is then able to highlight the problematic component in the graph when a service is degraded. The hierarchy of assurance graph helps to correlate a service degradation with the network root cause.

Fig. 2 illustrates the SAIN architecture. The *Service Configuration Orchestrator* is the system implementing the configurations to perform the service setup (i.e., it is an intent-based system). The SAIN *Orchestrator* retrieves the configurations of service instances and, then, convert them into an assurance graph. The SAIN *Agent* allows one to communicate with one or several devices (possibly also with another agent) in order to build the assurance graph by performing the necessary computations. An Agent also interact with monitored entities to retrieve metrics that will be later used to compute the health score. Finally, the SAIN *collector* retrieves the agents output to display it in a user friendly form.

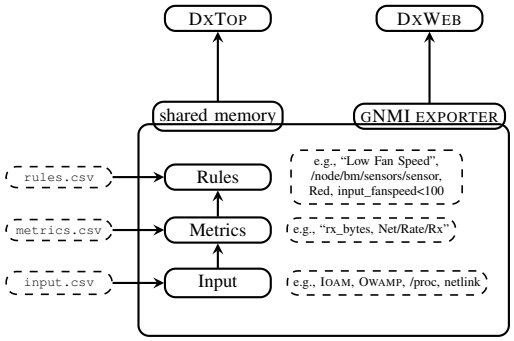



Fig. 3. DXAGENT architecture.

In this paper, we propose what is, to the best of our knowledge, the very first open source implementation of a SAIN Agent called *Diagnostic Agent* (DXAGENT) [12].

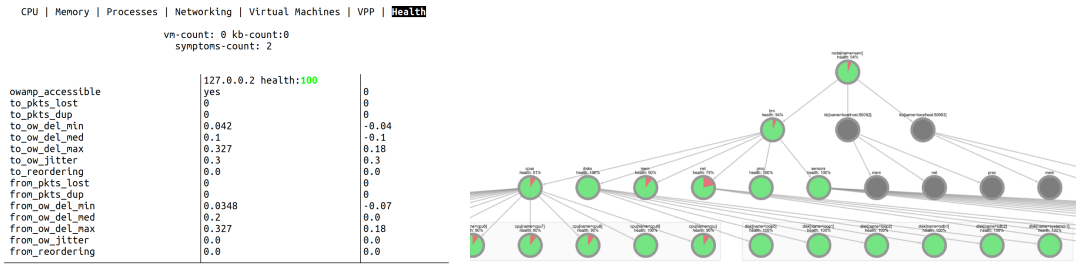
## III. DXAGENT

This section describes our implementation of the DXAGENT, as a SAIN Agent. It first provides the general overview of the DXAGENT (Sec. III-A) and, then, focuses on two particular telemetry data collection implementations: IOAM (Sec. III-B) and OWAMP (Sec. III-C).

### A. General Overview

The Diagnostic Agent (DXAGENT) is a SAIN Agent implementation in charge of computing the health scores by monitoring the devices useful to a service. The DXAGENT has been developed in Python and is freely available [16]. The overall architecture of the DXAGENT is illustrated in Fig. 3.

The DXAGENT operations are divided in three steps. First, the DXAGENT retrieves input from multiple sources. This corresponds to the `Input` module on Fig. 3. The data is collected directly on either the physical or virtual machine on which the DXAGENT is run but through telemetry measurements for VPP [17] and IOAM [14], [15] (see Sec. III-B and III-C). The data taken into account can be the number of packets received on an interface, the temperature of the computer components, the CPU usage, the current idle or sleeping process number, but also VirtualBox VMs and VPP [17] data. The `Input` module relies on a scheduler for harvesting data at a certain pace. By default, the `Input` module scheduler retrieves data every three seconds but this can be tuned according to needs and to a balance between data accuracy and device load. The file `input.csv` provides information on the type of data to collect and where it is available. With respect to the SAIN




(a) DXTOP– health tab opened. (b) DXWEB

Fig. 4. DXAGENT displaying results.

framework (see Fig. 2), the `Input` module corresponds to the metrics collection on monitored entities.

Once the data has been retrieved, it is possible to determine the set of subservices running on the machine or the network. For example, the list of active network interfaces is retrieved through the `Input` module, so we can determine that each interface is a subservice. In addition, inferring the dependencies between subservices allows the DXAGENT, through the `Metrics` module, to build the assurance graph. Note that, here, the DXAGENT differs from the SAIN general framework as the graph should be built and sent by the SAIN Orchestrator to the SAIN Agent (see Fig. 2). Thus, the `Input` and `Rules` modules allows the DXAGENT to act as the SAIN Orchestrator. Further, the `Metrics` module is also characterized by the normalization of the extracted data. It is important to gather data from different sources under the same variable and the same unit. For example, the number of bytes sent by an interface is retrieved from different sources depending on whether it is a physical or virtual machine. Since the third step is to create rules from the metrics, we only need one variable, in other words, a single metric that we could call `rx_bytes` for example. The normalization information is provided to the module by the file `metrics.csv`.

Finally, the `Rule` module checks for symptoms based on user-defined rules applied to normalized metrics, computes subservices health scores, and propagates health scores along the subservice assurance graph. Rules are provided to the module thanks to the file `rules.csv`. A typical rule is made of four elements (as illustrated in the dashed box next to the `Rule` module on Fig. 3): ($i$) the symptom name, ($ii$) the path to the subservice in the graph, ($iii$) the severity (two possible values: Red – health score decrease of 50% – and Orange – health score decrease of 10%), and ($iv$) the condition to be evaluated. For the example provided in Fig. 3, it means that a "Low Fan Speed" symptom will be triggered on a "sensor" subservice with a "Red" severity if the fan speed is below 100 RPM.

The DXAGENT is a daemon run in background and, as such, cannot be used as a user interface. Instead, it communicates with two other modules for displaying results. First, DXTOP, a console application, uses shared memory to communicate with the DXAGENT to get useful data to display, as illustrated in Fig. 4a. DXTOP proposes several tabs, most of them being dedicated to a given subservice. The last one, "Health", displays the symptoms of the monitored device.

Even if it is useful, the DXTOP does not offer a view of the assurance graph as a whole. Therefore, a graphical view of the assurance graph is provided by the DXWEB. It also shows the health scores and symptoms, as illustrated in Fig. 4b. The green portion of the nodes indicates the percentage of health of the subservice. It is possible to click on the nodes to know the symptoms of the subservices with a non-maximum health score.

With the DXTOP and DXWEB, the DXAGENT differs again from the general SAIN framework as both the DXTOP and DXWEB correspond to the SAIN Collector in (see Fig. 2).

Some network-related metrics provide insights about a node health but are not necessarily enough to accurately predict performance degradation of a specific service. By including the network forwarding path into these metrics, thanks to telemetry (see Sec. III-B and III-C), the DXAGENT is able to enhance the reported health and allows for a more precise issue detection. For example, one could retrieve the current size of a queue and see that it is almost full. However, it does not specify, for instance, which flows are impacted, why, since when, or for how long. With network telemetry, the DXAGENT gathers more data, therefore improving and clarifying the context of a problem.

### B. IOAM

The DXAGENT includes support for In-Situ OAM[2] (IOAM) [14], [15] with IPV6, which we have implemented in the Linux kernel. It is available [22], [16] since version 5.15. IOAM is considered as a hybrid OAM type that records telemetry data within packets. It is deployed in a given domain between the ingress and the egress or between selected devices within the domain. Each IOAM node on the path may insert or update IOAM data, e.g., node ID, interface IDs, timestamps, queue size, buffer occupancy. Having such telemetry data to enhance the network health is definitely a plus.

We have also implemented an IOAM agent [16] that collects IOAM data from packets and reports them. To retrieve such a data, both the DXAGENT and the IOAM

[2]Multiple *Operations, Administration, and Maintenance* (OAM) tools have been developed, for various layers in the protocol stack [18], going from basic `traceroute` to Bidirectional Forwarding Detection (BFD [19]) or recent UdpPinger [20] and Fbtracert [21]. The measurement techniques developed under the OAM framework have the potential for performing fault detection and isolation and for performance measurements.

Agent use `gNMI` [23] (`gRPC Network Management Interface`), i.e., a `gRPC` [10] wrapper for network-related stuff such as telemetry streaming. Once a connection is established between an IOAM agent and the DXAGENT, the latter fetches IOAM data every 10 seconds (it is totally tunable and can be changed), and so for each IOAM Agent it is connected to. The DXAGENT finally gathers and enhances the network health report with IOAM data freshly collected.

### C. OWAMP

The DXAGENT also includes an implementation of the One-Way Active Measurement protocol (OWAMP) [13]. OWAMP relies on a client/server architecture. While the protocol allows for many possibilities, the OWAMP implementation [24] can only measure the link between the client and the server. In particular, we focus on the server and the client implementing the ping utility. Doing so, one is now able to retrieve information about link characteristics (delay, loss, etc) at any time. OWAMP offers the possibility to ping a list of hosts on a recurring basis. This is achieved thanks to a tunable scheduler that is in charge of pinging every address of the list at a certain pace (e.g., one ping every 20 seconds).

We have developed a Python wrapper for integrating OWAMP into the DXAGENT [16]. The data retrieval is done by means of a file that will contain the ping OWAMP information. The callback function, contained in the DXAGENT, sent to the OWAMP scheduler will ask the scheduler to start writing the output to a file (one file per different addresses to ping).[3] In this way, when an input retrieval is scheduled by the DXAGENT, it is enough to read this file. Doing so, it is not necessary to synchronize both schedulers. Using any other method, such as shared memory or message passing, would not only complicate the procedure by adding synchronization concerns, but would be inconsistent with the way DXAGENT works.

## IV. USE CASES

In this section, we develop several use cases demonstrating how the integration of OWAMP with the DXAGENT works and how useful it can be. Sec. IV-A discusses our experimental methodology while Sec. IV-B illustrates our results.

### A. Methodology

We place ourselves in the context of a small network infrastructure offering, for instance, a video streaming service to its customers. The infrastructure, illustrated in Fig. 5 is an intent-based networking architecture and the streaming service has been set up using an intent. Our objective is to show that the DXAGENT can observe if an intent drift is present or about to appear by analyzing the network continuously.

In Fig. 5, the server (that could encompass a complete data center infrastructure) hosts the DXAGENT. It is thus from the server that measurements will be initiated and retrieved. Also,

[3]It is worth noticing this writing process is atomic, i.e., writing takes place in a temporary file and when the writing is finished, the temporary file is renamed by the file one wants to overwrite.

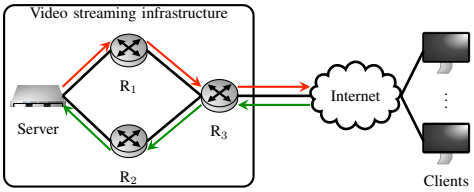



Fig. 5. Streaming service topology. The server runs the DXAGENT while the OWAMP server is running at $R_3$. Green arrows refer to server incoming packets. Red arrows refer to server outgoing packets.

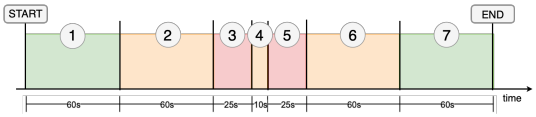



Fig. 6. Delay observation scenario. The whole experiment lasts 300. Router $R_x$ refers to router $R_1$ or $R_2$, depending on traffic direction.

the OWAMP server is running at router $R_3$ to analyze the traffic between this router and the server.

The intent that was proposed for the streaming video service is the following: "One wants a connection between the server and $R_3$ such that outgoing packets go through a different path than incoming packets and the one-way delay (whatever the direction) does not exceed 50 ms". This intent explains why outgoing traffic (red arrows on Fig. 5) goes through $R_1$, while incoming traffic (green arrows on Fig. 5) goes through $R_2$.

It is worth noticing that the intent shows a delay requirement only in the direction of packets leaving the server. This makes sense because the service provider wants to ensure that video packets are received quickly while the delay for client acknowledgment is potentially less important.

Finally, the OWAMP scheduler has been setup to three seconds, the packet size to 100 bytes, and the timeout to 0.5sec. The DXAGENT scheduler (i.e., the rate at which it updates its metrics) has been setup to the default value (i.e., three seconds). Finally, the OWAMP and DXAGENT scheduler, while they share the same value, are not synchronized and work independently of each other.

### B. Results

We consider three scenarios to illustrate the benefit of considering the DXAGENT with OWAMP. Sec. IV-B1 provides a scenario in which the DXAGENT observes and reacts to delays on paths. Sec. IV-B2 discusses a scenario in which packets are duplicated and reordered. Sec. IV-B3 focuses on link failure discovery.

*1) Observing Delay:* The first scenario aims at demonstrating that the DXAGENT reacts well to delay changes. This scenario is inspired by the intent used to build the streaming video infrastructure, i.e., outgoing one-way delay (whatever the direction) should not exceed 50 ms. The experiment scenario is provided in Fig. 6. States 1 and 7 are initial states in which the one-way delay between the server and $R_3$ (whatever the direction) is 2ms. States 2, 4, and 6 correspond to a situation in which the one-way delay is replaced by 50ms in both directions. Finally, States 3 and 4 refer to strongly degraded situations in which the one-delay is 100ms in both

```
Owamp delay > 100 : (from_ow_del_max+to_ow_del_max <= 200) and (from_ow_del_max+
      to_ow_del_max > 100)
Owamp rtt > 200 : from_ow_del_max+to_ow_del_max > 200
```

Fig. 7. DXAGENT rules for observing delay changes.

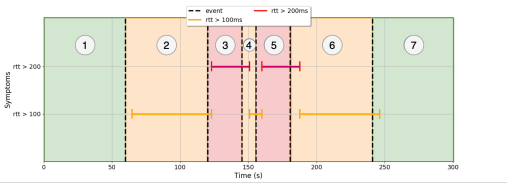



Fig. 8. Symptoms for the delay changes scenario.

```
5min(dynamicity(to_ow_del_med) + dynamicity(from_ow_del_med)) > 2
```

Fig. 9. Example of more complex rule for observing delay.

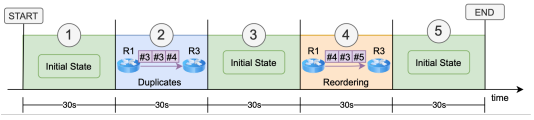



Fig. 10. Duplicates and reordering scenario. The whole experiment lasts 150sec.

directions. This scenario is motivated by the fact that the DXAGENT should able to determine a higher or lower delay rise and also to detect the return to a normal state which should be materialized by an absence of symptom. The DXAGENT rules (`Rules` module) for identifying those states are provided in Fig. 7.

It is simply a matter of returning a symptom of orange severity when the maximum RTT of a OWAMP ping exceeds 100 ms and of red severity if it exceeds 200ms. It is worth noticing that, for OWAMP, the RTT is the sum of the one-way delay in both path directions.

Fig. 8 shows the DXAGENT reaction with respect to the different states. We can see that symptoms are indeed observed when delay changes take place. Fig. 8 shows that the DXAGENT, thanks to the OWAMP input, is able to recognize the delay changes slightly after the beginning of the issue and it maintains information about the symptoms a little bit after the end of the issue (i.e., we are back to normal state). Those delays are due to the way we configure the DXAGENT scheduler, i.e., at which time interval the DXAGENT reads input file provided by OWAMP. This scheduler value could be, obviously, reduced to react more quickly to issues. But this would naturally lead to more processing from the DXAGENT perspective. It is thus up to the administrator to find the best balance between symptoms recognition and input readings.

It is worth noticing that the DXAGENT allows one to express more complex rules than the ones provided in Fig. 7. For instance, one may desire to observe a symptom in which the sum of the averages of the median one-way delays in both directions would be greater than two seconds for at least five minutes. This rule is illustrated in Fig. 9.

*2) Duplicates and Reordering:* The second scenario we want to test aims at determining whether the DXAGENT can

```
Owamp Duplication : to_pkts_dup >0
Owamp Reordering : to_reordering >0
```

Fig. 11. DXAGENT rules for identifying duplicates and reordering.

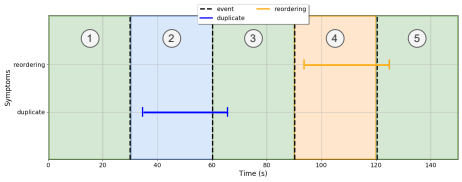



Fig. 12. Symptoms for the duplicates and reordering scenario.

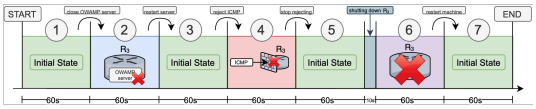



Fig. 13. Link failure scenario. The whole test lasts 430sec.

identify packets duplication or reordering. The experiment scenario is provided in Fig. 10. Each state in the experiment lasts 30 seconds. States 1, 3, and 5 corresponds to the initial state, where everything is working as expected (i.e., no reordering and no duplicates). One minute after the start of the experiment, every second packet from $R_1$ to $R_3$ is duplicated, corresponding to state 2. In addition, after the one minute and a half, every second packet from $R_1$ to $R_3$ is delayed by 100ms in order to create reordering. The DXAGENT rules (`Rules` module) for identifying duplicates and reordering are provided in Fig. 11.

These two rules ensure that if at least one packet is duplicated or reordered, the corresponding symptom will be reported by the DXAGENT. It is worth noticing that, $R_1$ being on the path from the server to the border router ($R_3$), it is the one-way duplicate and reordering of that direction that is observed.

Fig. 12 shows the DXAGENT reaction with respect to the different states. As expected in the scenario, packets duplicates appear after 30sec. Fig. 12 shows that the DXAGENT, thanks to the OWAMP input, is able to recognize the presence of duplication four seconds after the beginning of the issue and it maintains information about the symptoms until three seconds after the end of state 2 (i.e., we are back to normal state). The same applies for packets reordering (i.e., state 4).

*3) Link Failure Discovery:* The last scenario we test aims at demonstrating how the DXAGENT, thanks to OWAMP input, can detect link failures. The experiment scenario is provided in Fig. 13. Each state in the experiment lasts 60 seconds. States 1, 3, 5, and 7 are intended to serve as a reference state (i.e., everything is working as expected) between the various anomalies we will introduce into the network. The first link failure (state 2) corresponds to the shutdown of the OWAMP server located on $R_3$ before being turned on again 60 seconds later. The second anomaly (state 4) is to test the ICMP `ping` for 60 seconds by simply blocking the `echo_request` packets at the firewall. The last anomaly (state 6) is to shut down the container representing $R_3$. Shutting it down takes some time

```
Owamp - not reachable : owamp_accessible=="no"
Icmp - not reachable : icmp_accessible=="no"
```

Fig. 14. DXAGENT rules for identifying link failures.

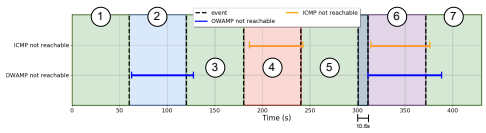



Fig. 15. Symptoms for the link failure discovery scenario.

```
Owamp - not reachable : owamp_accessible=="no" and icmp_accessible=="yes"
Icmp - not reachable : icmp_accessible=="no" and owamp_accessible=="yes"
Machine not reachable : owamp_accessible=="no" and icmp_accessible=="no"
```

Fig. 16. Improved DXAGENT rules for identifying link failures.

(close to ten seconds); the real time has been calculated and is visible on Fig. 15. The DXAGENT rules (`Rules` module) for identifying those link failures are provided in Fig. 14.

Fig. 15 show the results for the second experiment. As expected, when the OWAMP server is off (State 2), OWAMP pings do not reach their destination while classic ICMP `ping` does not encounter any problems. But when the `echo_request` is blocked (State 4), it is the opposite. Finally, the most interesting case to analyze is when the router is turned off (State 6). First of all and fortunately, neither of the two pings reach their destination. This result is interesting because by multiplying the sources we can establish a better vision of the network that we want to diagnose.

To better discriminate between OWAMP server and router issues, we could improve the DXAGENT rules as illustrated in Fig. 16. Of course, the fact that the machine is not accessible for both protocols does not necessarily mean that the machine is off, but we are getting closer to the truth than with only one source. Moreover, we notice that OWAMP observes more quickly that the machine has been turned off (State 6) and takes more time to realize that it has started up compared to traditional pings. This additional delay corresponds to the time required for the server to boot, on the contrary to the classic `ping` that is part of the TCP/IP stack.

## V. CONCLUSION

This paper introduced the Diagnostic Agent (DXAGENT), a SAIN Agent implementation for service assurance. Our DXAGENT collects data both vertically (i.e., on the machine itself) but also horizontally (i.e., telemetry data obtained with OWAMP and IOAM) and, then, discovers the various subservices to build an assurance graph. Based on the retrieved data, the DXAGENT checks for potential symptoms and spreads the corresponding health scores on the assurance graph. Use cases have demonstrated how useful is the DXAGENT. All our code is freely available.

The DXAGENT described in this paper is the first building block towards closed-loop automation for service assurance. In the near future, we plan to investigate how machine learning and artificial intelligence may help in automatically correlating metrics with the DXAGENT assurance graph but also possibly in automatically infer initial rules for symptoms. Also, an important step towards closed loop is to ensure that symptoms are not seen as another log message. One possibility would be to improve rules and symptoms such that it contains a pointer towards the data model information.

## REFERENCES


[1] A. Clemm, L. Ciavaglia, L. Granville, and J. Tantsura, "Intent-based networking – concepts and definitions," Internet Engineering Task Force, Internet Draft (Work in Progress) draft-irtf-nmrg-ibn-concepts-definitions-06, December 2021.

[2] E. Zeydan and Y. Turk, "Recent advances in intent-based networking: A survey," in *Proc. IEEE Vehicular Technology Conference (VTC)*, May 2020.

[3] B. K. Saha, D. Tandur, L. Haab, and L. Podleski, "Intent-based networks: An industrial perspective," in *Proc. International Workshop on Fugure Industrial Communication Networks*, October 2018.

[4] Global Market Insights, "Intent-based networking market," April 2020, see https://www.gminsights.com/industry-analysis/intent-based-networking-ibn-market (last access: Feb. 17th, 2022).

[5] P. Gill, M. Arlitt, Z. Li, and A. Mahant, "The flattening Internet topology: Natural evolution, unsightly barnacles or contrived collapse?" in *Proc. Passive and Active Measurement Conference (PAM)*, April 2008.

[6] A. Dhamdhere and C. Dovrolis, "The Internet is flat: Modeling the transition from a transit hierarchy to a peering mesh," in *Proc. ACM CoNEXT*, December 2010.

[7] H. Zhao and J. Bi, "Characterizing and analysis of the flattening Internet topology," in *Proc. International Symposium on Computers and Communications (ISCC)*, July 2013.

[8] T. Böttger, F. Cuadrado, G. Tyson, I. Castro, and S. Uhlig, "Open connect everywhere: A glimpse at the Internet ecosystem through the lens of the netflix CDN," *ACM SIGCOMM Computer Communication Review*, vol. 48, no. 1, January 2018.

[9] A. Clemm and E. Voit, "Subscription to YANG notifications for datastore updates," Internet Engineering Task Force, RFC 8641, September 2019.

[10] gRPC, "A high performance, open source universal RPC framework," see grpc.io (Last Access: Feb. 23rd, 2022).

[11] B. Claise, J. Quilbeuf, D. Lopez, D. Voyer, and T. Arumugam, "Service assurance for intent-based networking architecture," Internet Engineering Task Force, Internet Draft (Work in Progress) draft-ietf-opsawg-service-assurance-architecture-02, October 2021.

[12] K. Edeline, "Diagnostic agent," 2021, see https://github.com/Advanced-Observability/dxagent.

[13] S. Shalunov, B. Teitelbaum, A. Karp, J. Boote, and M. Zekauskas, "A one-way active measurement protocol (OWAMP)," Internet Engineering Task Force, RFC 4656, September 2006.

[14] F. Brockners, S. Bhandari, and T. Mizrahi, "Data fields for in-situ OAM," Internet Engineering Task Force, Internet Draft (Work in Progress) draft-ietf-ippm-ioam-data-17, December 2021.

[15] S. Bhandari, F. Brockners, C. Pignataro, H. Gredler, J. Leddy, S. Youell, T. Mizrahi, A. Kfir, B. Gafni, P. Lapukhov, M. Spiegel, S. Krishnan, R. Asati, and M. Smith, "In-situ OAM ipv6 options," Internet Engineering Task Force, Internet Draft (Work in Progress) draft-ietf-ippm-ioam-ipv6-options-07, February 2022.

[16] J. Iurman, K. Edeline, T. Carlisi, and B. Donnet, "Advanced observability," 2021, see https://github.com/Advanced-Observability.

[17] "VPP technology," FD.io Vector Packet Processing, see https://fd.io/gettingstarted/technology/.

[18] T. Mizrahi, N. Sprecher, E. Bellagamba, and Y. Weingarten, "An overview of operations, administration, and maintenance (OAM) tools," Internet Engineering Task Force, RFC 7276, June 2014.

[19] D. Katz and D. Ward, "Bidirectional forwarding detection (BFD)," Internet Engineering Task Force, RFC 5880, June 2010.

[20] Facebook, "Udppinger," see https://github.com/facebook/UdpPinger.

[21] ——, "fbtracert," see https://github.com/facebook/fbtracert.

[22] J. Iurman, "Support for the IOAM pre-allocated trace with IPv6," *LWN.net*, July 2021, see https://lwn.net/Articles/863746/.

[23] Cisco Innovation Edge, "cisco-gnmi-python," see https://github.com/cisco-ie/cisco-gnmi-python (Last Access: Feb. 23rd, 2022).

[24] "OWAMP implementation," see https://github.com/perfsonar/owamp.